COMSC-210 Lecture Topic 13
n-squared Sorting Algorithms

Reference
Childs Ch. 14
sorting-algorithms.com

Sorting Paradigm
"selection sort"
  scan all elements
"insertion sort"
  scan only elements you need to scan

Considerations
filled or not -- go up to "fill line"
"holes" or not -- skip not inUse

Nested For-Loop Sort
basic sorting algorithm for arrays
  in any language
use when no other way is possible

```
for (i = 0; i < n; i++) // stop at n
{
  if (!a[i].inUse) continue; // in case of holes
  for (j = i + 1; j < n; j++)
  {
    if (!a[j].inUse) continue; // in case of holes
    if (a[j] < a[i]) // for LO-to-HI sorting
    {
      T temp = a[i];
      a[i] = a[j];
      a[j] = temp;
    }
  }
}
```

"bubble-sort" is a variation
HI-to-LO sorting: `if (a[i] < a[j])`

Using A Swap Function
C++ STL's "algorithm" library has `swap(T&, T&)`

```
for (i = 0; i < n; i++) // stop at n
{
  if (!a[i].inUse) continue; // in case of holes
  for (j = i + 1; j < n; j++)
  {
    if (!a[j].inUse) continue; // in case of holes
    if (a[j] < a[i]) swap(a[i], a[j]);
  }
}
```

e.g., Python: `a[i],a[j] = a[j],a[i]`

Linked List Sorting

```
for (Node* p = start; p != 0; p = p->next)
{
  if (!p->inUse) continue; // in case of holes
  for (Node* q = p->next; q != 0; q = q->next)
  {
    if (!q->inUse) continue; // in case of holes
    if (q->value < p->value)
    {
      swap(*p, *q); // swap EVERYTHING
      swap(p->next, q->next); // swap pointer back
    }
  }
}
```

The Insertion Sort
inserts values, moving to make space
click here for code sample
Analysis: #of comparisons: size^2 / 4
about same timing complexity as nested for-loop, but...
  best case is O(n)

The Shell Sort: Variation Of Insertion Sort
attempts to minimize swaps vs insertion sort
click here for code sample
Analysis: #of moves: size$^{1.25}$ (that is, O(n$^{1.25}$))
...*does not adapt well to linked lists!*
  because `i += d` does not translate well

C++ Library Sort Function
STL's sort function (ref)
  in the `algorithm` library
  e.g., `sort (myvector.begin(), myvector.end());`

C's qsort Function (ref)
  in the `cstdlib` library
  variant of partition-exchange sorting

Binary Search
given an ordered array:
  divide into 2 halves
  choose half that should contain match value
  repeat until size of half is one
big oh: O(log n)

Binary Search Algorithm for Arrays

Binary Search Timing Complexity

"a" is an array of type DataType
1st parameter:
"s" is the starting index of the array (usually 0)
2nd parameter:
"e" is the index AFTER the last
index to be included in the search (usually size)
3rd parameter:
"value" is the DataType value to be matched

LOOP STARTS HERE

```
int m = (s + e - 1) / 2; // the middle element
if (value == am) // got lucky -- matches middle element
  return m; // return index of found element
else if (s == e - 1) // 1-element array
  return -1; // only element in array did not match
else if (value < am) // look between s and m-1
{
  if (s == m) return -1; // but that range is empty, no match
  else {e = m; continue;} // look in s to m-1
}
else // look between m+1 and e-1
{
  if (m == e - 1) return -1; // but that range is empty, no match
  else {s = m + 1; continue;} // look in m-1 to e
}
```

LOOP ENDS HERE

To find a specific value in an ordered array of 8 values (that is, n = 8), divide the 8 into two groups of 4 each. If the value is between the two groups, the search is unsuccessful and completed. Otherwise, determine which group of 4 might contain the value, and discard the other group of 4 each.

Divide the 4 into two groups of 2. If the value is between the two groups, the search is unsuccessful and completed. Otherwise, determine which group of 2 might contain the value, and discard the other group of 2 each.

Divide the 2 into two groups of 1 each. See if the value is in either of the two groups, or not.

This takes 3 steps of operations to divide and check two groups. To apply this to an initial array of 16, add just one more step to divide into groups of 8. To apply this to an initial array of 32, add just one more step to divide into groups of 16, and so on.

It looks like this:

n	#of steps
8	3
16	4
32	5

...or $2^{\#of\ steps} = n$.
...or `#of steps = lg(n)`, where lg is $\log_2$.

Since log base 10 equals log base e time a constant, equals log base 2 times a constant, time is proportional to log(n). (That is, log(n) =

log(2) * lg(n). Also FWIW, log(n) = log(e) * ln(n).) Hence, binary search is O(log(n)), or more commonly written as **O(log n)**.